Scheduling Parallel Tasks: Approximation Algorithms

Pierre-Francois Dutot, Grégory Mounié, Denis Trystram

Scheduling Parallel Tasks Approximation Algorithms

Pierre-François Dutot, Grégory Mounié and Denis Trystram
ID-IMAG
51 avenue Jean Kuntzmann
38330 Montbonnot Saint Martin, France

August 1, 2003

Abstract

Scheduling is a crucial problem in parallel and distributed processing. It consists in determining where and when the tasks of parallel programs will be executed. The design of parallel algorithms has to be reconsidered by the influence of new execution supports (namely, clusters of workstations, grid computing and global computing) which are characterized by a larger number of heterogeneous processors, often organized by hierarchical sub-systems.

Parallel Tasks model (tasks that require more than one processor for their execution) has been introduced about 15 years ago as a promising alternative for scheduling parallel applications, especially in the case of slow communication media. The basic idea is to consider the application at a rough level of granularity (larger tasks in order to decrease the relative weight of communications). As the main difficulty for scheduling in actual systems comes from handling efficiently the communications, this new view of the problem allows to consider them implicitely, thus leading to more tractable problems.

We kindly invite the reader to look at the chapter of Maciej Drozdowski (in this book) for a detailed presentation of various kinds of Parallel Tasks in a general context and the survey paper from Feitelson et al. [14] for a discussion in the field of parallel processing. Even if the basic problem of scheduling Parallel Tasks remains NP-hard, some approximation algorithms can be designed. A lot of results have been derived recently for scheduling the different types of Parallel Tasks, namely, Rigid, Moldable or Malleable ones. We will distinguish Parallel Tasks inside a same application or between applications in a multi-user context. Various optimization criteria will be discussed.

This chapter aims to present several approximation algorithms for scheduling moldable and malleable tasks with a special emphasis on new execution supports.

1 Introduction: Parallel Tasks in Parallel Processing

1.1 Motivation

As it is reflected in this book, Scheduling is a very old problem which motivated a lot of researches in many fields. In the Parallel Processing area, this problem is a crucial issue for determining the starting times of the tasks and the processor locations. Many theoretical studies were conducted [3, 32, 7] and some efficient practical tools have been developed (Pyrros [18], Hypertool [46]).

Scheduling in modern parallel and distributed systems is much more difficult because of new characteristics of these systems. These last few years, super-computers have been replaced by collections of large number of standard components, physically far from each other and heterogeneous [10]. The needs of efficient algorithms for managing these resources is a crucial issue for a more popular use. Today, the lack of adequate software tools is the main obstacle for using these powerful systems in order to solve large and complex actual applications.

The classical scheduling algorithms that have been developed for parallel machines of the nineties are not well adapted to new execution supports. The most important factor is the influence of communications. The first attempts that took into account the communications into computational models were to adapt and refine existing models into more realistic ones (delay model with unitary delays [23], LogP model [9, 26]). However, even the most elementary problems are already intractable [43], especially for large communication delays (the problem of scheduling simple bipartite graphs is already NP-hard [2]).

1.2 Discussion about Parallel Tasks

The idea behind Parallel Tasks is to consider an alternative for dealing with communications, especially in the case of large delays. For many applications, the developers or users have a good knowledge of their behavior. This

qualitative knowledge is often enough to guide the parallelization.

Informally, a Parallel Task (PT) is a *task* that gathers elementary operations, typically a numerical routine or a nested loop, which contains itself enough parallelism to be executed by more than one processor. This view is more general than the standard case and contains the sequential tasks as a particular case. Thus, the problems of scheduling PT are at least as difficult to solve. We can distinguish two ways for building Parallel Tasks:

- PT as parts of a large parallel application. Usually off-line analysis is possible as the time of each routine can be estimated quite precisely (number of operations, volume of communications), with precedence between PT.
- PT as independent jobs (applications) in a multi-user context. Usually, new PT are submitted at any time (on-line). The time for each PT can be estimated or not (clairvoyant or not) depending on the type of applications.

The PT model is particularly well-adapted to grid and global computing because of the intrinsic characteristics of these new types of supports: large communication delays which are considered implicitely and not explicitely like they are in all standard models, the hierarchical character of the execution support which can be naturally expressed in PT model and the capacity to react to disturbances or to imprecise values of the input parameters. The heterogeneity of computational units or communication links can also be considered by uniform or unrelated processors for instance.

1.3 Typology of Parallel Tasks

There exist several versions of Parallel Tasks depending of their execution on a parallel and distributed system (see [13] and Drozdowski's chapter of this book).

- *Rigid* when the number of processors to execute the PT is fixed a priori. This number can either be a power of 2 or any integer number. In this case, the PT can be represented as a rectangle in a Gantt chart. The allocation problem corresponds to a strip-packing problem [28].
- *Moldable* when the number of processors to execute the PT is not fixed but determined before the execution. As in the previous case this number does not change until the completion of the PT.

- In the most general case, the number of processors may change during the execution (by preemption of the tasks or simply by data redistributions). In this case, the Parallel Tasks are *Malleable*.

Practically, most parallel applications are moldable. An application developer does not know in advance the exact number of processors which will be used at run time. Moreover, this number may vary with the input problem size or number of nodes availability. This is also true for many numerical parallel library. Most of the main restrictions are the minimum number of processors that will be used because of time, memory or storage constraints. Some algorithms are also restricted to particular data sizes and distributions like the FFT algorithm where 2^q processors are needed or Strassen's matrix multiplication with its decomposition into 7^q subproblems [16].

Most parallel programming tools or languages have some malleability support, with dynamic addition of processing nodes support. This is already the case since the beginning for the well-known message passing language PVM, where nodes can be dynamically added or removed. This is also true from MPI-2 libraries. It should be noticed that an even more advanced management support exists when a Client/Server model is available like in CORBA, RPC (remote procedure call) or even MPI-2 [41]. Modern advanced academic environments, like Condor, Mosix, Cilk, Satin, Jade, NESL, PM^2 or Athapascan implement very advanced capabilities, like resilience, preemption, migration, or at least the model allows to implement these features.

Nevertheless, most of the time moldability or malleability must still be taken explicitly into account by the application designers as computing power will appear, move or be removed. This is easy in a Master/Worker scheme but may be much more difficult in a SPMD scheme where data must then be redistributed. Environments abstracting nodes may, theoretically, manage these points automatically.

The main restriction in the moldability use is the need for efficient scheduling algorithm to estimate (at least roughly) the parallel execution time in function of the number of processors. The user has this knowledge most of the time but this is an inertia factor against the more systematic use of such models.

Malleability is much more easily useable from the scheduling point of view but requires advanced capabilities from the runtime environment, and thus restrict the use of such environments and their associated programming models. In the near future, moldability and malleability should be used more

and more.

1.4 Tasks Graphs

We will discuss briefly in this section how to obtain and handle Parallel Tasks.

We will consider two types of Parallel Tasks corresponding repectively to independent jobs and to applications composed by large tasks.

The purpose of this chapter is not to detail and discuss the representation of applications as graphs for the sake of parallelization. It is well-known that obtaining a symbolic object from any application implemented in a high-level programming language is difficult. The graph formalism is convenient and may be declined in several ways. Generally, the coding of an application can be represented by a directed acyclic graph where the vertices are the instructions and the edges are the data dependencies [8].

In a typical application, such a graph is composed by tens, hundreds or thousands of tasks, or even more. Using symbolic representation such as in [25], the graph may be managed at compile time. Otherwise, the graph must be build on-line, at least partially, at the execution phase. A moldable, or malleable, task graph is a way to gather elementary sequential tasks which will be handled more easily as it is much smaller.

There are two main ways to build a task graph of Parallel Tasks: either the user has a relatively good knowledge of its application and is able to provide the graph (top-down approach), or the graph is built automatically from a larger graph of sequential tasks generated at run-time (down-top).

1.5 Content of the chapter

In the next section, we introduce all the important definitions and notations that will be used throughout this chapter. The central problem is formally defined and some complexity results are recalled.

We then start with a very simple case, to introduce some of the methods that are used in approximation algorithms. The following six sections are oriented on some interesting problems covering most of the possible combinations which have been studied in the literature and have been resolved with very different techniques.

The sections are ordered in the following way:

- The criterion is the most important. We start with C_{max}, then $\sum C_i$ and finally both (see next section for definitions).
- The off-line versions are discussed before the on-line versions.
- Moldable Tasks are studied before Malleable Tasks.
- Finally precedence constraints: from the simple case of independent tasks to more complicated versions.

This order is related to the difficulty of the problems. For example off-line problems are simpler than on-line problems.

Finally we conclude with a discussion on how to consider other characteristics of new parallel and distributed systems.

2 Formal Definition and Theoretical Analysis

Let us first introduce informally the problem to solve: Given a set of Parallel Tasks, we want to determine at what time the tasks will start their execution on a set of processors such that at any time no more than m processors are used.

2.1 Notations

We will use in this chapter standard notations used in the other chapters of the book.

Unless explicitely specified, we consider n tasks executed on m identical processors.

The execution time is denoted $p_j(q)$ when task j (for $1 \leq j \leq n$) is allocated to q processors. The starting time of task j is $\sigma(j)$, and its completion time is $C_j = \sigma(j) + p_j(q)$. When needed, the number q of processors used by task j will be given by $q = nbproc(j)$.

The *work* of task j on q processors is defined as $w_j(q) = q \times p_j(q)$. It corresponds to the surface of the task on the Gantt chart (time-space diagram).

We will restrict the analysis on Parallel Tasks that start their execution on all processors simultaneously. In other words, the execution of rigid tasks or moldable tasks corresponds to rectangles. The execution of malleable tasks corresponds to an union of contiguous rectangles.

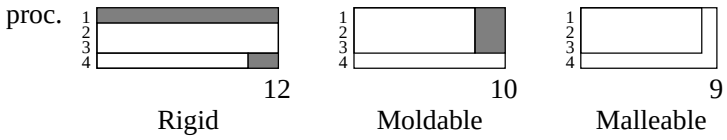

Figure 1: Comparison of the execution of rigid, moldable and malleable tasks.

The following figure represents the execution of two Parallel Tasks in the three contexts of rigid, moldable and malleable tasks.

The tasks used in this figure have the execution times presented in table 1 for the moldable case. For the malleable case, percentage of these times are taken.

Table 1: Tasks used in figure 1

Number of processors	1	2	3	4
Big task	24	12	8	7
Small task	10	9	6	5

In the rigid case, the big task can only be executed on two processors and the small task on one processor. In the moldable case, the scheduler can choose the allocation but cannot change it during the execution. Finally in the malleable case, the allocation of the small task is on one processor for 80% of its execution time and on all the four processors for the remaining 20%.

As we can see, moldable and malleable characters may improve the rigid execution.

2.2 Formulation of the problem

Let us consider an application represented by a precedence task graph $G(V, E)$. The PTS (Parallel Tasks Schedule) problem is defined as follows:
Instance: A graph $G = (V, E)$ of order n, a set of integer $p_j(q)$ for $1 \leq q \leq m$ and $1 \leq j \leq n$.
Question: Determine a feasible schedule which minimizes the objective function f. We will discuss in the next section of the different functions used in the literature.

A feasible schedule is a pair of functions $(\sigma, nbproc)$ of $V \rightarrow N \times [1..m]$, such as:

- The precedence constraints are verified: $\sigma(j) \geq \sigma(i) + p_i(nbproc(i))$ if task j is a successor of i (there is no communication cost),
- At any time slot no more than m processors are used.

2.3 Criteria

The main objective function used historically is the *makespan*. This function measures the ending time of the schedule i.e. the latest completion time over all the tasks. However, this criterion is valid only if we consider the tasks all together and from the viewpoint of a single user. If the tasks have been submitted by several users, other criteria can be considered. Let us review briefly the different possible criteria usually used in the literature:

- Minimisation of the *makespan* (completion time $C_{max} = max(C_j)$ where C_j is equal to $\sigma(j) + p_j(nbproc(j))$)
- Minimisation of the average completion time (ΣC_i) [37, 1] and its variant weighted completion time ($\Sigma \omega_i C_i$). Such a weight may allow to distinguish some tasks from each other (priority for the smallest ones, etc.).
- Minimisation of the mean stretch (defined as the sum of the difference between release times and completion times). In an on-line context it represents the average response time between the submission and the completion.
- Minimisation of the maximum stretch (i.e. the longest waiting time for a user).
- Minimisation of the tardiness. Each task is associated to an expected due date and the schedule must minimise either the number of late tasks, the sum of the tardiness or the maximum tardiness.
- Other criteria may include rejection of tasks or normalized versions (with respect to the workload) of the previous ones.

In this chapter, we will focus on the first two criteria which are the most studied.

2.4 Performance ratio

Considering any previous criteria, we can compare two schedules on the same instance. But the comparison of scheduling algorithms requires an additional metric. The performance ratio is one of the standard tool used to compare the quality of the solutions computed by scheduling algorithms [21].

It is defined as follows: The performance ratio ρ_A of algorithm A is the maximum over all instances $\mathcal{I}$ of the ratio $\frac{f(\mathcal{I})}{f^*(\mathcal{I})}$ where f is any minimization criterion and f^* is the optimal value.

Throughout the text, we will use the same notation for optimal values. The performance ratios are either constant or may depend on some instance input data like the number of processors, tasks or precedence relation.

Most of the time the optimal values could not be computed in reasonable time unless $P = NP$. Sometimes, the worst case instances and values may not be computed neither. In order to do the comparison, approximation of these values are used. For correctness, a lower bound of the optimal value and an upper bound of the worst case value are computed in such cases.

Some studies also use the mean performance ratio, which is better than the worst case ratio, either with a mathematical analysis or by empirical experiments.

Another important feature for the comparison of algorithms is their complexities. As most scheduling problems are NP-Hard, algorithms for practical problems compute approximate solutions. In some contexts, algorithms with larger performance ratio may be prefered thanks to their lower complexity, instead of algorithms providing better solutions but at a much greater computational cost.

2.5 Penalty and monotony

The idea of using Parallel Tasks instead of sequential ones was motivated by two reasons, namely to increase the granularity of the tasks in order to obtain a better balance between computations and slow communications, and to hide the complexity of managing explicit communications.

In the Parallel Tasks model, communications are considered as a global *penalty* factor which reflects the overhead for data distributions, synchronization, preemption or any extra factors coming from the management of the parallel execution. The penalty factor implicitely takes into account some

constraints, when they are unknown or too difficult to estimate formally. It can be determined by empirical or theoretical studies (benchmarking, profiling, performance evaluation through modeling or measuring, etc.).

The penalty factor reflects both influences of the Operating System and the algorithmic side of the application to be parallelized.

In some algorithms, we will use the following hypothesis which is common in the parallel application context. Adding more processors usually reduces the execution time, at least until a threshold. But the speedup is not super-linear. From the application point of view, increasing the number of processors also increases the overhead: more communications, more data distributions, longer synchronizations and termination detection, etc.

Hypothesis 1 (Monotony) *For all tasks j, p_j and w_j are monotonic:*

- *$p_j(q)$ is a decreasing function in q*
- *$w_j(q)$ is an increasing function in q*

More precisely,

$$p_j(q+1) \leq p_j(q)$$

and

$$w_j(q) \leq w_j(q+1) = (q+1)p_j(q+1) \leq (1+\frac{1}{q})q\ p_j(q) = (1+\frac{1}{q})w_j(q)$$

.

Figure 2 gives a geometric interpretation of this hypothesis.

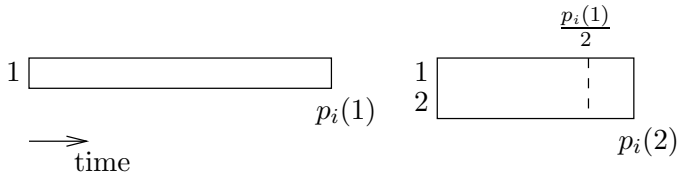

Figure 2: Geometric interpretation of the penalty on 2 processors.

From the parallel computing point of view, this hypothesis may be interpreted by the Brent's lemma [6]: if the instance size is large enough, a parallel execution should not have super-linear speedup. Sometimes, parallel applications with memory hierarchy cache effect, race condition on flow control, or scheduling anomalies described by Graham [19], may lead to such

super-linear speedups. Nevertheless, most parallel application fulfill this hypothesis as their performances are dominated by communication overhead.

Other general hypotheses will be considered over this chapter, unless explicitely stated:

- A processor executes at most one task at a time.
- Preemption between Parallel Tasks is not allowed (but preemption inside PT can be considered, in this case, its cost will be included as part of the penalty). A task can not be stopped and then resumed, or restarted. Nevertheless the performance ratio may be established in regard to the preemptive optimal solution.

2.6 Complexity

Table 2 presents a synthetic view of the main complexity results linked with the problems we are considering in this chapter. The rigid case has been deeply studied in the survey [11].

All the complexity proof for the rigid case involving only sequential tasks can be extended to the moldable case and to the malleable case with a penalty factor which does not change the execution time on any number of processors. All the problems of table 2 are NP-Hard in the strong sense.

Table 2: NP-Hard problems and associated reductions

problem			reduction
C_{max}	Indep.	off-line	from 3-partition
		on-line clairvoyant	from the off-line case
		on-line non-clairvoyant	from the off-line case
	Prec.	off-line	from $P\|p_i = 1, prec\|C_{max}$
$\sum \omega_i C_i$	Indep.	off-line	from $P\|\|\sum \omega_i C_i$
		on-line	from the off-line case

3 Preliminary analysis of a simplified case

Let us first detail a specific result on a very simple case. Minimizing C_{max} for identical moldable tasks is one of the simplest problem involving moldable tasks. This problem has some practical interest as many applications

generate at each step a set of identical tasks to be computed on a parallel platform.

With precedence constraints this problem is as hard as the classical problem of scheduling precedence constrained unit execution time tasks (UET) on multiprocessors, as a moldable task can be designed to run with the same execution time on any number of processors.

Even without precedence constraints, there is no known polynomial optimal scheduling algorithm and the complexity is still open. To simplify even more the problem, we introduce a phase constraint. A set of tasks is called a *phase* when all the tasks in the set start at the same time, and no other task starts executing on the parallel platform before the completion of all the tasks in the phase.

This constraint is very practical as a schedule where all the tasks are run in phases is easier to implement on a actual system.

For identical tasks, if we add the restriction that tasks are run in phases, the problem becomes polynomial for simple precedence graph like trees. When the phases algorithm is used for approximating the general problem of independent tasks, the performance ratio is exactly 5/4.

3.1 Dominance

When considering such a problem, it is interesting to establish some properties that will restrict the search for an optimal schedule. With the phase constraint, we have one such property.

Proposition 1 *For a given phase length, the maximum number of tasks in the phase is reached if all the tasks are alloted to the same number of processors, and the number of idle processors is less than this allocation.*

The proof is rather simple, let us consider a phase with a given number of tasks. Within these tasks, let us select one of the tasks which are the longest. This task can be chosen among the longest as one with the smallest allocation. There is no task with a smaller allocation than the selected one because the tasks are monotonic.

This task runs in less than the phase length. All other tasks are starting at the same time as this task. If the tasks with a bigger allocation are given the same allocation as the selected one, they will all have there allocation reduced (therefore this transformation is possible). The fact that their running time will probably increase is not a problem here as we said that within a phase all the tasks are starting simultaneously. Therefore it is possible to

change any phase in a phase where all tasks have the same allocation. The maximum number of tasks is reached if there is not enough idle processors to add another task.

3.2 Exact resolution by dynamic programming

Finding an optimal phase by phase schedule is a matter of splitting the number n of tasks to be scheduled into a set of phases which will be run in any order. As the number of tasks in a phase is an integer, we can solve this problem in polynomial time using integer dynamic programing. The principle of dynamic programming is to say that for one task the optimal schedule is one phase of one task, for two tasks the optimal is either one phase of two tasks or one phase of one task plus the optimal schedule of one task and so on.

The makespan ($C_{max}(n)$) of the computed schedule for n tasks is:

$$C_{max}(n) = \min_{i=1..m} \left(C_{max}(n-i) + p_j \left(\left\lfloor \frac{m}{i} \right\rfloor \right) \right)$$

The complexity of the algorithm is in $O(mn)$.

3.3 Approximation of the relaxed version

We may think that scheduling identical tasks in a phase by phase schedule produces the optimal result even for the problem where this phase by phase constraint is not imposed. Indeed there is a great number of special cases where this is true. However there are some counter examples as in figure 3. This example is built on five processors, with moldable tasks running in 6 units of time on one processor, 3 units of time on two processors and 2 units of time on either three, four and five processors.

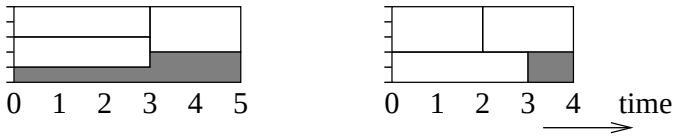

Figure 3: The optimal schedules with and without phases for 3 moldable tasks on 5 processors.

This example shows that the performance ratio reached by the phase algorithm is greater than $\frac{5}{4}$. To prove that it is exactly $\frac{5}{4}$, we need to make

a simple but tedious and technical case analysis on the number of tasks (see [40] for the details).

4 Independent Moldable Tasks, Cmax, off-line

In this section, we focus on the scheduling problem itself. We have chosen to present several results obtained for the same problem using different technics.

Let us consider the scheduling of a set of n independent moldable tasks on m identical processors for minimizing the makespan. Most of the existing methods for solving this problem have a common geometrical approach by transforming the problem into 2 dimensional packing problems. It is natural to decompose the problem in two successive phases: determining the number of processors for executing the tasks, then solve the corresponding problem of scheduling rigid tasks.

The next section will discuss the dominance of the geometrical approach.

4.1 Discussion about the geometrical view of the problem

We discuss here the optimality of the rectangle packing problem in scheduling moldable tasks. The figure below shows an example of non contiguous allocation in the optimal schedule. Moreover, we prove that no contiguous allocation reaches the optimal makespan in this example.

The instance is composed by the eight tasks given in table 3, to be executed on a parallel machine with 4 processors.

Table 3: Execution times of the 8 tasks of figure 4

Tasks	1	2	3	4	5	6	7	8
1 proc.	13	18	20	22	6	6	12	3
2 proc.	13	18	20	22	3	3	6	1.5
3 proc.	13	18	20	22	2	3	6	1
4 proc.	13	18	20	22	2	3	6	1

Proof is left to the reader that these tasks verify the monotony assumptions.

The minimum total workload (sum of the first line) divided by the number of processors gives a simple lower bound for the optimal makespan. This optimal value is reached with the schedule presented in figure 4, where task 8 is allocated to processors a, c and d.

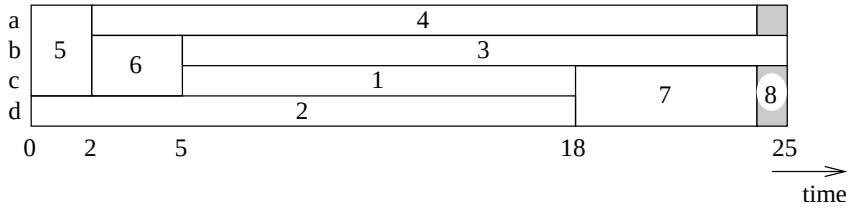

Figure 4: Optimal non contiguous allocation.

We will now prove that no contiguous allocation can have a makespan of 25. One can easily verify that no permutation of the four processors in figure 4 gives a contiguous allocation.

First, let us look at the possible allocations without considering if a schedule is feasible or not.

Given the sizes of tasks 1 to 4, we cannot allocate two of these tasks to a processor. Therefore let us say that task 1 is allocated to processor 1, task 2 to processor 2 and so on. The idle time left on the processors is 12 units of time for processor 1, 7 on processor 2, 5 for processor 3 and 3 for processor 4.

Task 7 being the biggest of the remaining tasks, it is a good starting point for a case study. If task 7 is done sequentially, it can only be alloted on processor 1 and leaves no idle time on this processor. In this case, we have processors 2, 3, and 4 with respectively 7, 5 and 3 units of idle time. The only way to fill the idle time of processor 3 is to put task 5 on all three processors and task 6 on two processors. With the allocation of task 5 we have 5, 3 and 1 units of idle time, and with allocation of task 6 we have 2, 0 and 1 units of idle time. Task 8 cannot be allocated to fill two units of time on processor 2 and one on processor 4. Therefore the assumption that task 7 can be done sequentially is wrong.

If task 7 cannot be done sequentially, it has to be done on processor 1 and 2 as these are the only ones with enough idle time. This leaves respectively 6, 1, 5 and 3 units of idle time. The only way to fill the processor 2 is to allocate task 8 on three processors. Which leaves either 5, 4 and 3 or 5, 5 and 2 or 6, 4 and 2. With only task 5 and 6 remaining, the only possibility to perfectly fit every task in place is to put task 5 on three processors and task 6 on 2 processors.

The resulting allocation is shown in figure 5. On this figure, no scheduling has been made. The tasks are just represented on the processors they are allotted to, according with the previous discussion. The numbers on the left are the processors indices and the letters on the right show that we can

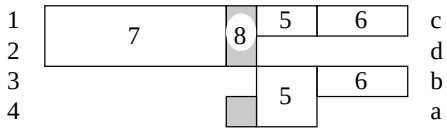

Figure 5: The resulting allocation.

relate each processor to one from figure 4. The only possible allocation is the one used in the schedule of figure 4. As we said that no permutation of the processors can give a contiguous representation of this allotment, there is no contiguous scheduling in 25 units of time.

4.2 Two phases approach

We present in this section a first approximation algorithm for scheduling independent moldable tasks using a two phases approach: first determining the number of processors for executing the tasks, then, solving the corresponding rigid problem by a strip packing algorithm.

The idea that has been introduced in [44] is to optimize in the first phase the criterion used to evaluate the performance ratio of the second phase. The authors proposed to realize a trade-off between the maximum execution time (critical path) and the sum of the works.

The following algorithm gives the principle of the first phase. A more complicated and smarter version is given in the original work.

Compute an allocation with minimum work for every task
while $\frac{\sum w_j(nbproc(j))}{m} < max_j(p_j(nbproc(j)))$ **do**
  Select the task with the largest execution time
  Change its allocation for another one with a strictly smaller execution time and the smallest work
end while

After the allocation has been determined, the rigid scheduling may be achieved by any algorithm with a performance ratio function of the critical path and the sum of the works. For example a strip-packing algorithm like Steinberg's one [39] fulfills all conditions with an absolute performance ratio of 2. A recent survey of such algorithms may be found in [28]. Nevertheless if contiguity is not mandatory, a simple rigid multiprocessor list scheduling algorithm like [17] reaches the same performance ratio of 2. We will detail this algorithm in the following paragraph. It is an adaptation of the classical

Graham's list scheduling under resource constraints.

The basic version of the list scheduling algorithm is to schedule the tasks using a list of priority executing a rigid task as soon as enough resources are available. The important point is to schedule the tasks at the earliest starting time, and if more than one task is candidate to consider first the one with the highest priority. In the original work, each task being executed use some resources. The total number of resources is fixed and each task requires a specific part of these resources. In the case of rigid independent tasks scheduling, there is only one resource which corresponds to the number of processors allocated to each task and no more than m processors may be used simultaneously. In Graham's paper, there is a proof for obtaining a performance ratio of 2 which can be adapted to our case.

Proposition 2 *The performance ratio of the previous algorithm is 2.*

The main argument of the proof is that the trade-off achieved by this algorithm is the best possible, and thus the algorithm is better in the first phase than the optimal scheduling. The makespan is driven by the performance ratio of the second phase (2 in the case of Steinberg's strip packing).

The advantage of this scheme is its independence in regard to any hypothesis on the execution time function of the tasks(like monotony). The major drawback is the relative difficulty of the rigid scheduling problem which constrains here the moldable scheduling. In the next sections, we will take another point of view: put more emphasis on the first phase in order to simplify the rigid scheduling on the second phase.

It should be noticed that none of the strip packing algorithms explicitly use in the scheduling the fact that the processor dimension is discrete. We present such an algorithm in section 4.3 with a better performance ratio of only $3/2+\epsilon$ for independent moldable tasks with the monotony assumption.

4.3 A better approximation

The performance ratio of Turek's algorithm is fixed by the corresponding strip packing algorithm (or whatever rigid scheduling algorithm used). As such problems are NP-hard, the only way to obtain better results is to solve different allocation problems which lead to "easier" scheduling problems.

The idea is to determine the task allocation with great care in order to fit them into a particular packing scheme. We present below a 2-shelves algorithm [29] with an example on figure 6.

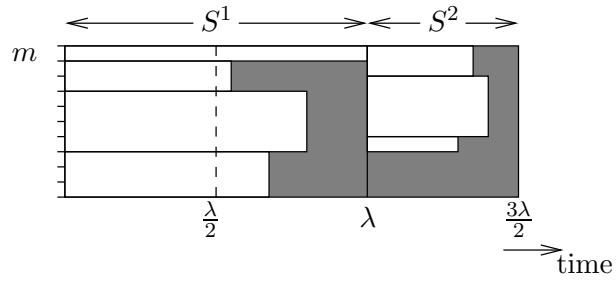

Figure 6: Principle of the 2 shelves allocation S^1 and S^2.

This algorithm has a performance ratio of $3/2 + \epsilon$. It is obtained by stacking two shelves of respective sizes λ et $\frac{\lambda}{2}$ where *lambda* is a guess of the optimal value C^*_{max}. This guess is computed by a dual approximation scheme [22]. Informally, the idea behind dual approximation is to fix an hypothetical value for the guess λ and to check if it is lower than the optimal value C^*_{max} by running a heuristic with a performance ratio equal to ρ and a value C_{max}. If $\lambda < \frac{1}{\rho}C_{max}$, by definition of the performance ratio, λ is underestimated. A binary search allows to refine the guess with an arbitrary accuracy ϵ.

The guess λ is used to bound some parameters on the tasks. We give below some constraints that are useful for proving the performance ratio. In the optimal solution, assuming $C^*_{max} = \lambda$:

- $\forall j$, $p_j(nbproc(j)) \leq \lambda$.
- $\sum w_j(nbproc(j)) \leq \lambda m$.
- When two tasks share the same processor, the execution of one of these tasks is lower than $\frac{\lambda}{2}$. As there are no more than m processors, less than m processors are used by the tasks with an execution time larger than $\frac{\lambda}{2}$.

We will now detail how to fill the two shelves S^1 and S^2 (figure 6), as best as we can with respect to the sum of the works. Every tasks in a shelf start at the beginning of the shelf and are allocated on the minimum number of processors to fit into. The shelf S^2 (of length lower than $\lambda/2$) may be

overfilled, but the sum of processors used for executing the tasks in the first shelf S^1 is imposed to be less than m. Hopefully, this last problem can be solved in polynomial time by dynamic programming with a complexity in $O(nm)$. We detail below the corresponding algorithm.

---

define $\gamma(j,d)$:= minimum nbproc(j) such that $p_j(nbproc(j)) \leq d$
$W_{0,0} = 0; W_{\forall j,q<0} = +\infty;$
for $j = 1..n$ **do**
  for $q = 1..m$ **do**

$$W_{j,q} = min \begin{pmatrix} W_{j-1,q-\gamma(j,\lambda)} + W_{j,\gamma(j,\lambda)} & // \text{ in } S^1 \\ W_{j-1,q} + W_{j,\gamma(j,\lambda/2)} & // \text{ in } S^2 \end{pmatrix}$$

  end for
end for

The sum of the works is smaller than λm (otherwise the λ parameter of the dual approximation scheme is underestimated and the guess must be changed).

Let us now build a feasible schedule. All tasks with an allocation of 1 (sequential tasks) and an execution time smaller than $\lambda/2$ will be put away and considered at the end.

The goal is to ensure that most processors compute for at least λ time units, until all the tasks fit directly into the two shelves.

All tasks scheduled in S^2 are parallel ($nbproc(j) \geq 2$) and, according to the monotony assumption, have an execution time greater than $\lambda/4$. We have to ensure that tasks in S^1 use more than $3\lambda/4$ of processing power. While S^2 is overfilled, we do some technical changes among the following ones:

- stack two sequential tasks ($nbproc(j) = 1$) in S^1 with an execution time smaller than $3\lambda/4$, and schedule them sequentially on a single processor.
- decrease $nbproc(j)$ of one processor for a parallel task in S^1 whose execution time is smaller than $3\lambda/4$ and schedule it alone on $nbproc(j)-1$ processors.
- schedule one task from S^2 in S^1 without overfilling it, changing the task allocation to get a execution time smaller than λ.

The two first transformations use particular processors to schedule one or two "large" tasks, and liberate some processors in S^1. All transformations decrease the sum of the works. A surface argument shows that the third

transformation occurs at most one time and then the scheduling becomes feasible. Up to this moment, one of the previous transformations is possible.

The small sequential tasks that have been removed at the beginning fit between the two shelves without increasing C_{max} more than $3\lambda/2$ because the total sum of the works is smaller than λm (in this case, at least always one processor is used less than λ units of time).

---

for Dichotomy over $\lambda : \sum w_j(1)/m \leq \lambda \leq \sum w_j(m)/m$ **do**
  $small = tasksj : p_{j,1} \leq \lambda/2$
  $large = remainingtasks$
  knapsack for selecting the tasks in S^1 and S^2
  if $\sum w_j(nbproc(j)) > \lambda m$ **then**
    Failed, increase λ
  else
    Build a feasible schedule for $large$
    insert $small$ between the two shelves
    Succeed, decrease λ
  end if
end for

Proposition 3 *The performance ratio of the 2-shelves algorithm is* $\frac{3}{2} + \epsilon$

The proof is quite technical, but it is closely linked with the construction. It is based on the following surface argument: the total work remains always lower than the guess λm. Details can be found in [30].

4.4 Linear Programming approach

There exists a polynomial time scheme for scheduling moldable independent tasks [24]. This scheme is not fully polynomial as the problem is NP-Hard in the strong sense: the complexity is not polynomial in regard to the chosen performance ratio.

The idea is to schedule only the tasks with a "large" execution time. All combinations of the tasks with all allocations and all orders are tested. At the end, the remaining small tasks are added. The important point is to keep the number of "large" tasks small enough in order to keep a polynomial time for the algorithm.

The principle of the algorithm is presented below.

---

for $j = 1..n$ **do**
  $d_j = min_{l=1..m} p_j(l)$

end for
$D = \sum_{j=1}^{n} d_j$
$\mu = \epsilon/2m$
$K = 4m^{m+1}(2m)^{\lceil 1/\mu \rceil + 1}$
$k = min_{k \leq K}(d_k + ... + d_{2mk+3m^{m+1}-1} \leq \mu D)$
Construct the set of all the relative order schedules involving the k tasks with the largest d_j (denoted by $\mathcal{L}$)
for $R \in \mathcal{L}$ **do**
Solve (approximately) R mappings, using linear programming
Build a feasible schedule including remaining tasks.
end for
return := the best built scheduling.

We give now the corresponding linear program for a particular element of $\mathcal{L}$. A relative order schedule is a list of g snapshots $M(i)$ (not to be mistaken with M the set of available processors). A snapshot is a subset of tasks and the processors where they are executed. $P(i)$ is the set of processors used by snapshot $M(i)$. $\mathcal{F} = \{M \backslash P(i), i = 1..g\}$, i.e. the set of free processors in every snapshot. $P_{F,i}$ is one of the n_F partition of $F \in \mathcal{F}$. For each partition the number of processor sets F_h, with cardinality l is denoted $a_l(F, i)$. A task appears in successive snapshots, from snapshot α_i to snapshot ω_i. D^l is the total processing time for all tasks not in $\mathcal{L}$ (denoted S). Note that solutions are non integer, thus the solution is postprocessed in order to build a feasible schedule.

Minimize t_g s.t.

1. $t_0 = 0$
2. $t_i \geq t_{i-1}, i = 1..g$
3. $t_{w_j} - t_{\alpha_j - 1} = p_j, \forall T_j \in \mathcal{L}$
4. $\sum_{i:P(i)=M\backslash F}(t_i - t_{i-1}) = e_F, \forall F \in \mathcal{F}$
5. $\sum_{i=1}^{n_F} x_{F,i} \leq e_F, \forall F \in \mathcal{F}$
6. $\sum_{F \in \mathcal{F}} \sum_{i=1}^{n_F} a_l(F, i) x_{F,i} \geq D^l, l = 1..m$
7. $x_{F,i} \geq 0, \forall F \in \mathcal{F}, i = 1..n_F$
8. $\sum_{T_j \in S} t_j(l) y_{jl} \leq D^l, l = 1..m$
9. $\sum_{l=1}^{m} y_{jl} = 1, \forall T_j \in S$

10. $y_{jl} \geq 0, \forall T_j \in S, l = 1..m$

where t_i are snapshot end time. the starting time t_0 is 0 and the makespan t_g. e_F is the time while processors in F are free. $x_{F,i}$ the total processing time for $P_{F,i} \in \mathcal{P}_F, i = 1..n_F, F \in \mathcal{F}$ where only processors of F are executing short tasks and each subset of processors $F_j \in P_{F,i}$ executes at most one short task at each time step in parallel. The last three constraints defined moldable allocation. In the integer linear program, y_{jl} is equal to 1 if task T_j is allocated to l processors, 0 otherwise.

The main problem is to solve a linear program for every $(2^{m+2}k^2)^k$ allocations and orders of the k tasks. This algorithm is of little practical interest, even for small instances and a large performance ratio. Actual implementations would prefer algorithms with a lower complexity like the previous algorithm with a performance ratio of 3/2.

5 General Moldable Tasks, Cmax, off-line

5.1 Moldable Tasks with precedence

Scheduling Parallel Tasks that are linked by precedence relations corresponds to the parallelization of applications composed by large modules (library routines, nested loops, etc.) that can themselves be parallelized.

In this section, we give an approximation algorithm for scheduling any precedence task graph of moldable tasks. We consider again the monotonic hypothesis. We will establish a constant performance ratio in the general case.

The problem of scheduling moldable tasks linked by precedence constraints has been considered under very restricted hypotheses like those presented in section 5.2.

Another way is to use a direct approach like in the case of the two-phases algorithms. The monotony assumption allows to control the allocation changes. As in the case of independent moldable tasks, we are looking for an allocation which realizes a good trade-off between the sum of the works (denoted by W) and the critical path (denoted by T_∞).

Then, the allocation of tasks is changed in order to simplify the scheduling problem. The idea is to force the tasks to be executed on less than a fraction of m, eg. $m/2$. A task does not increase its execution time more than the inverse of this fraction thanks to the monotony assumption. Thus,

the critical path of the associated rigid task graph does not increase more than the inverse of this fraction.

With a generalisation of the analysis of Graham [19], any list scheduling algorithm will fill more than half (i.e. 1 - the fraction) of the processors at any time, otherwise, at least one task of every paths of the graph is being executed. Thus, the cumulative time when less than $m/2$ processors are occupied is smaller than the critical path. As in first hand, the algorithm doubles the value of the critical path, and in second hand the processors work more than $m/2$ during at most $2W/m$, the overall guaranty is $2W/m+2T_\infty$, leading to a performance ratio of 4.

Let us explain in more details how to choose the ratio. With a smart choice [27] a better performance ratio than 4 may be achieved. The idea is to use three types of time intervals, depending on if the processors are used more or less than μ and $m-\mu$ (see I_1, I_2 and I_3 in figure 7. For the sake of clarity, the intervals have been represented as contiguous ones). The intervals I_2 and I_3 where tasks are using less than $m-\mu$ processors are bounded by the value of the critical path, and the sum of the works bounds the surface corresponding to intervals I_1 and I_2 where more than μ processors are used. The best performance ratio is reached for a value of parameter μ depending on m, with $1 \leq \mu \leq m/2+1$, such that:

$$r(m) = min_\mu max \left\{ \frac{m}{\mu}, \frac{2m-\mu}{m-\mu+1} \right\}$$

We can now state the main result.

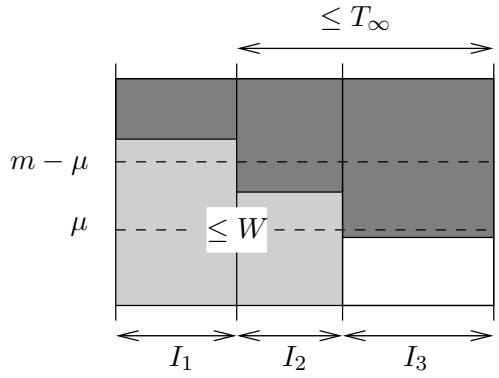

Figure 7: The different time interval types.

Proposition 4 *Performance ratio of the previous algorithm is* $\frac{3+\sqrt{5}}{2}$ *for serie-parallel graphs and trees.*

The reason of the limitation of the performance ratio is the ability to compute an allocation which minimizes the critical path and sum of the works. An optimal solution may be achieved for structured graphs like trees using dynamic programming with a deterministic and reasonable time. In the general case, it is still possible to choose an allocation with a performance ratio of 2 in regard to the critical path and sum of the works. The overall performance ratio is then doubled (that is $3 + \sqrt{5}$).

5.2 Relaxation of continuous algorithms

We have presented in section 4 some ways to deal with the problem of scheduling independent moldable tasks for minimizing the makespan. The first two approaches considered direct constructions of algorithms with a small complexity and reasonable performance ratios, and the last one used a relaxation of a continuous linear program with a heavy complexity for a better performance. It is possible to obtain other approximations from a relaxation of continuous resources (i.e. where a parallel task may be allocated to a fractional number of processors).

Several studies have been done for scheduling precedence task graphs.

- Prasanna et al. [33] studied the scheduling of graphs where all tasks have the same penalty with continuous allocations. The speed-up functions (which are inversely proportional to the penalty factors) are restricted to values of type q^{α}, where q is the fraction of the processors allocated to the tasks and $0 \leq alpha \leq 1$. This hypothesis is stronger than the monotony assumption and is far from the practical conditions in Parallel Processing.

- Using restricted shapes of penalty functions (concave and convex), [45] provided optimal execution schemes for continuous Parallel Tasks. For concave penalty factors, we retrieve the classical result of the optimal execution in gang for super-linear speed-ups.

- Another related work considered the folding of the execution of rigid tasks on a smaller number of processors than specified [15]. As this folding has a cost, it corresponds in fact to the opposite of the monotony assumption, with super-linear speed-up functions. Again, this hypothesis is not practically realistic for most parallel applications.

- A direct relaxation of continuous strategies has been used for the case of independent moldable tasks [5] under the monotony assumption. This work demonstrated the limitations of such approaches. The continuous and discrete execution times may be far away from each other, e.g. for tasks which require a number of processors lower than 1. If a task requires a continuous allocation between 1 and 2, there is a rounding problem which may multiply the discrete times by a factor of 2. Even if some theoretical approximation bounds can be established, such an approach has intrinsic limitations which did not show any advantage over ad-hoc discrete solutions like those described in section 4. However, they may be very simple to implement!

6 Independent Moldable Tasks, Cmax, on-line batch

An important characteristic of the new parallel and distributed systems is the versatility of the resources: at any moment, some processors (or groups of processors) can be added or removed. On another side, the increasing availability of the clusters or collections of clusters involved new kind of data intensive applications (like data mining) whose characteristics are that the computations depend on the data sets. The scheduling algorithm has to be able react step by step to arrival of new tasks and thus, off-line strategies can not be used. Depending on the applications, we distinguish two types of on-line algorithms, namely, clairvoyant on-line algorithms when most parameters of the Parallel Tasks are known as soon as they arrive, and non-clairvoyant ones when only a partial knowledge of these parameters is available. We invite the readers to look at the survey of Sgall [36] or the chapter of the same author in this book.

Most of the studies about on-line scheduling concern independent tasks, and more precisely the management of parallel resources. In this section, we consider only the clairvoyant case, where a good estimate of the task execution time is known.

We present first a generic result for batch scheduling. In this context, the tasks are gathered into sets (called batches) that are scheduled together. All further arriving tasks are delayed to be considered in the next batch. This is a nice way for dealing with on-line algorithms by a succession of off-line problems. We detail below the result of Shmoys et al. [38] which proposed how to adapt an algorithm for scheduling independent tasks without release dates (all tasks are available at date 0) with a performance ratio of ρ into a

batch scheduling algorithm with unknown release dates with a performance ratio of 2ρ.

Figure 8 gives the principle of the batch execution and illustrates the notations used in the proof.

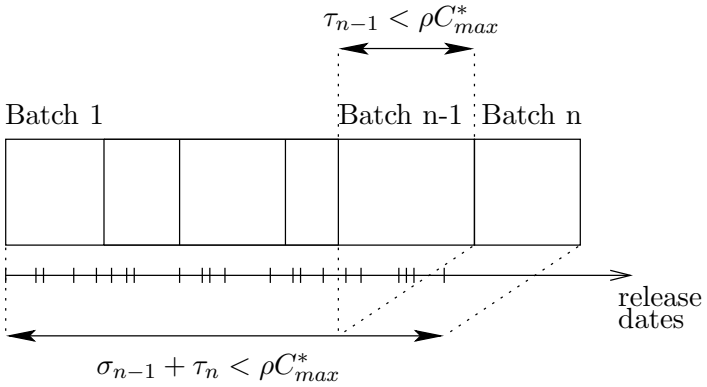

Figure 8: On-line schedule with batches.

The proof of the performance ratio is simple. First, let us remark that for any instance, the on-line optimal makespan is greater than the off-line optimal makespan. By construction of the algorithm, every batch schedules a subset of the tasks, thus every batch execution time τ_k is smaller than ρ times the optimal off-line makespan.

The last previous last batch starts before the last release date of a task. Let σ_{n-1} be the starting time of this batch. In addition, all the tasks in the last batch are also scheduled after σ_{n-1} in the optimal. Let τ_n be the execution time of the last batch. As the part of the optimal schedule after the time instant σ_{n-1} contains at least all the tasks of the last batch, the length l of this part times ρ is greater than τ_n. Therefore $\sigma_{n-1} + \tau_n < \sigma_{n-1} + \rho l < \rho C^*_{max}$.

If we consider the total time of our schedule as the sum of the time of the last previous last batch (τ_{n-1}) and the time of all other batches ($\sigma_{n-1} + \tau_n$), the makespan is clearly lower than $2\rho C^*_{max}$.

Now, using the algorithm of section 4.3 with a performance ratio of $3/2 + \epsilon$, it is possible to schedule moldable independant tasks with release dates with a performance ratio of $3 + \epsilon$ for C_{max}. The algorithm is a batch scheduling algorithm, using the independent tasks algorithm at every phase.

7 Independent Malleable Tasks, Cmax, on-line

Even without knowing tasks execution times, when malleable tasks are ideally parallel it is possible to get the optimal competitive ratio of $1 + \phi \approx 2.6180$ with the following deterministic algorithm [15]:

---

if an available task i requests $nbproc(j)$ processors and $nbproc(j)$ processors are available **then**
  schedule the task on the processors
end if
if less than m/ϕ processors are busy and some task is available **then**
  schedule the task on all available processors, folding its execution
end if

We consider in this section a particular class of non clairvoyant Parallel tasks which is important in practice in the context of exploitation of parallel clusters for some applications [4]: the expected completion times of the Parallel Tasks is unknown until completion (it depends on the input data), but the qualitative parallel behaviour can be estimated. In other words, the p_j are unknown, but the penalty functions are known.

We will present in this section a generic algorithm which has been introduced in [34] and generalized in [42]. The strategy uses a restricted model of malleable tasks which allows two types of execution, namely sequential and rigid. The execution can switch from one mode to the other. This simplified hypothesis allows to establish some approximation bounds and is a first step towards the general malleable case.

7.1 A generic execution scheme

We consider the on-line execution of a set of independent malleable tasks whose p_j are unknown. The number of processors needed for the execution of j is fixed (it will be denoted by q_j). The tasks may arrive at any time, but they are executed by successive batches. We assume that j can be scheduled either on 1 or q_j processors and the execution can be preempted.

We propose a generic framework based on batch scheduling. The basic idea is simple: when the number of tasks is large, the best way is to allocate the tasks to processors without idle times and communications. When enough tasks have been completed, we switch to a second phase with (rigid) Parallel Tasks. In the following analysis, we assume that in the first phase each job is assigned to one processor, thus, working with full efficiency. Inefficiency appears when less than m tasks remain to be executed. Then, when

the number of idle processors becomes larger than a fixed parameter α, all remaining jobs are preempted and another strategy is applied in order to avoid too many idle times. Figure 9 illustrates the principle of an execution of this algorithm. Three successive phases are distinguished: first phase when all the processors are busy; second phase when at least $m - \alpha + 1$ processors work (both phases use the well-known Graham's list scheduling); final phase when α or more processors become idle, and hence turn to a second strategy with Parallel Tasks.

Remark that many strategies can be used for executing the tasks in the last phase. We will restrict the analysis to rigid tasks.

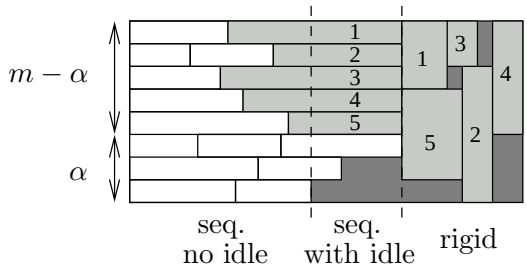

Figure 9: Principle of the generic scheme for partial malleable tasks.

7.2 Analysis

We provide now a brief analysis of the generic algorithm with idle regulation. More details can be found in [42].

Proposition 5 *The performance ratio of the generic scheme is bounded by:*

$$\frac{2m - q_{max}}{m - q_{max} + 1} - \alpha(\frac{1}{m - q_{max} + 1} - \frac{1}{m})$$

where q_{max} is the maximum of the q_j.

The proof is obtained by bounding the time in each phase. Let us check now that the previous bound corresponds to existing ones for some specific cases:

- The case $\alpha = 0$ corresponds to schedule only rigid jobs by a list algorithm. This strategy corresponds to a 2 dimensional packing problem which has already been studied in this chapter.
- The case $\alpha = 1$ for a specific allocation of processors in gang in the final phase (i.e. where each task is allocated to the full machine: $q_{max} = m$) has been studied in [34].
- The case $\alpha = m$ corresponds simply to list scheduling for sequential tasks (the algorithm is restricted to the first phase). As $q_{max} = 1$, the bound becomes $2 - 1/m$.

It is difficult to provide a theoretical analysis for the general case of malleable tasks (preemption at any time for any number of processors). However, many strategies can be imagined and implemented. For instance, if the penalties are high, we can switch progressively from the sequential execution to two, then three (and so on) processors. If the tasks are very parallel ones, it is better to switch directly from 1 to a large number of processors.

8 Independent Moldable Tasks, $\sum C_i$, off-line

In this section, we come back to the original problem of scheduling independent moldable Parallel Tasks focusing on the minimization of the other criterion, namely, the average completion time.

For a first view of the problem, we present two lower bounds and the principle of the algorithm from Schwiegelshohn et al. [35] which is dedicated to this criterion.

In this section we will use i instead of j for indexing the tasks because of the classical notation of $\sum C_i$.

8.1 Lower bounds

With this criterion, there is a need for new lower bounds instead of the ones generally used with the makespan: critical path and sum of the works.

A first lower bound is obtained when all the tasks start at time 0. The tasks complete no sooner than their execution times which depend on their allotment. Thus $H = \sum p_i(nbproc(i))$ is a lower bound of $\sum C_i$ for a particular allotment.

The second lower bound is obtained when considering the minimum work for executing all the tasks. From classical single processor scheduling, the

optimal solution for $\sum C_i$ is obtained by scheduling the tasks by increasing size order. Combining both arguments and assuming that each task may use m processors without increasing its area, we obtained a new lower bound when the tasks are sorted by increasing area: $A = \frac{1}{m}\sum w_i(nbproc(i))(n - i + 1)$.

This last bound is refined by the authors, using $W = \frac{1}{m}\sum w_i(nbproc(i))$ and a continuous integration. Like in the article, to simplify the notation, we present the original equation for rigid allocations. The uncompleted ratio of task i at time t is defined as

$$r(i) = \begin{cases} 1 & \text{if } t \leq \sigma(i) \\ 1 - \frac{t-\sigma(i)}{p_i} & \text{if } \sigma(i) \leq t \leq C_i \\ 0 & \text{if } C_i \leq t \end{cases}$$

Thus,

$$\sum_{i=1}^{n} \int_0^{+\infty} r_i(t) = \sum_{i=1}^{n} \int_0^{C_i} 1dt - \int_{\sigma(i)}^{C_i} \frac{t - \sigma(i)}{p_i}$$

As $C_i = \sigma(i) + p_i$ and $\int_{\sigma(i)}^{C_i} \frac{t-\sigma(i)}{p_i} = \frac{p_i}{2}$, it can be simplified as

$$\sum_{i=1}^{n} \int_0^{+\infty} r_i(t) = \sum_{i=1}^{n} C_i - \frac{1}{2}H$$

This result holds also for a transformation of the instance where the tasks keep the same area but use m processors. For this particular instance, gang scheduling by increasing height is optimal thus $\sum_{i=1}^{n} C'_i = A$ and $H' = W'$. As A is a lower bound of $\sum_{i=1}^{n} C_i$ and $H > H' = W' = W$:

$$\sum_{i=1}^{n} C_i - \frac{1}{2}H \geq A - \frac{1}{2}W$$

namely

$$\sum_{i=1}^{n} C_i \geq A + \frac{1}{2}H - \frac{1}{2}W$$

The extension to moldable tasks is simple: these bounds behave like the critical path and the sum of the works for the makespan. When H decreases, $A + \frac{1}{2}H - \frac{1}{2}W$ increases. Thus, there exists an allotment minimizing the maximum of both lower bounds. It can be used for the rigid scheduling.

8.2 Scheduling algorithm

We do not detail too much the algorithm as an algorithm with a better performance ratio is presented in section 9.

The "smart SMART" algorithm of [35] is a shelf algorithm. It has a performance ratio of 8 in the unweighted case and 8.53 in the weighted case ($\sum \omega_i C_i$). All shelves have a height of 2^k (1.65^k in the weighted case). All tasks are bin-packed (first fit, largest area first) into one of the shelves just sufficient to include it. Then all shelves are sorted in order to minimize $\sum C_i$, using a priority of $\frac{H_l}{\sum_l \omega_i}$, where H_l is the height of shelf l.

The basic point of the proof is that the shelves may be partitioned in two sets: a set including exactly one shelf of each size, and another one including the remaining shelves. Their completion times are respectively bounded by H and by A. The combination can be adjusted to get the best performance ratio (leading to the value of 1.65 in the weighted case).

9 Independent Moldable Tasks, bi-criterion, on-line batch

Up to now, we only analyzed algorithms with respect to one criterion. We have seen in section 2.3 that several criteria could be used to describe the quality of a scheduling. The choice of which criterion to choose depends on the priorities of the users.

However, one could wish to get the advantage of several criteria in a single scheduling. With the makespan and the sum of weighted completion times, it is easy to find examples where there is no schedule reaching the optimal value for both criteria. Therefore you can not have the cake and eat it, but you can still try to find for a schedule how far the solution is from the optimal one for each criterion. In this section, we will look at a generic way design algorithms with guaranties on two criteria and at a more specific algorithm family for the moldable case.

9.1 Two phases, two algorithms ($\mathcal{A}_{\sum C_i}$, $\mathcal{A}_{C_{max}}$)

Let us use two known algorithms $\mathcal{A}_{\sum C_i}$ and $\mathcal{A}_{C_{max}}$ with performance ratios respectively $\rho_{\sum C_i}$ and $\rho_{C_{max}}$ with respect to the sum of completion time and the makespan [31].

Proposition 6 *It is possible to combine $\mathcal{A}_{\sum C_i}$ and $\mathcal{A}_{C_{max}}$ in a new algorithm with a performance ratio of $2\rho_{\sum C_i}$ and $2\rho_{C_{max}}$ at the same time.*

Let us remark that delaying by τ the starting time of the tasks of the schedule given by $\mathcal{A}_{C_{max}}$ increases the completion time of the tasks with the same delay τ.

The starting point of the new algorithm is the schedule built by $\mathcal{A}_{\sum C_i}$. The tasks ending in this schedule before $\rho_{C_{max}} C^*_{max}$ are left unchanged. All tasks ending after $\rho_{C_{max}} C^*_{max}$ are removed and rescheduled with $\mathcal{A}_{C_{max}}$, starting at $\rho_{C_{max}} C^*_{max}$ (see figure 10). As $\mathcal{A}_{C_{max}}$ is able to schedule all tasks in $\rho_{C_{max}} C^*_{max}$ and it is always possible to remove tasks from a schedule without increasing its completion time, all these tasks will complete before $2\rho_{C_{max}} C^*_{max}$.

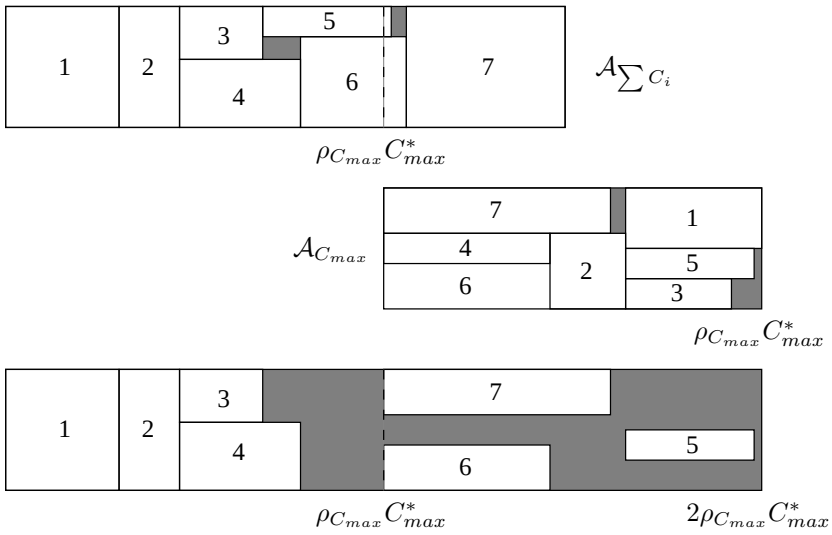

Figure 10: Bi-criterion scheduling combining two algorithms.

Now let us look at the new values of the two criteria. Any task scheduled by $\mathcal{A}_{\sum C_i}$ ending after $\rho_{C_{max}} C^*_{max}$ does not increase its completion time by a factor more than 2, thus the new performance ratio is no more than twice $\rho_{\sum C_i}$. On the other side, the makespan is lower than $2\rho_{C_{max}} C^*_{max}$. Thus the performance ratios on the two criteria are the double of the performance ratio of each single algorithm.

We can also remark that in figure 10 the schedule presented has a lot of idle times and the makespan can be greatly improved by just starting every tasks as soon as possible with the same allocation and order. However, even if this trick can give very good results for practical problems, it does not improve the theoretical bounds proven on the schedules, as it cannot always

be precisely defined.

Tuning performance ratios

It is possible to decrease one performance ratio at the expense of the other. The point is to choose a border proportionally to $\rho_{C_{max}} C^*_{max}$, namely $\lambda * \rho_{C_{max}} C^*_{max}$. The performance ratios are a Pareto curve of λ.

Proposition 7 *It is possible to combine $\mathcal{A}_{\sum C_i}$ and $\mathcal{A}_{C_{max}}$ in a new algorithm with a performance ratio of $\frac{1+\lambda}{\lambda}\rho_{\sum C_i}$ and $(1+\lambda)\rho_{C_{max}}$ at the same time.*

Combining the algorithm of section 4.3 and 8 it is possible to schedule independent moldable tasks with a performance ratio of 3 for the makespan and 16 for the sum of the completion time.

9.2 Multiple phases, one algorithm ($\mathcal{A}_{C_{max}}$)

The former approach required the use of two algorithms, one per criterion and mixed them in order to design a bi-criterion scheduling. It is also possible to design an efficient bi-criterion algorithm just adapting an algorithm $\mathcal{A}_{C_{max}}$ designed for the makespan criterion [20].

The main idea is to create a schedule which has a performance ratio on the sum of completion times based on the result of algorithm $\mathcal{A}_{C_{max}}$ without losing too much on the makespan. To have this performance ratio $\rho_{\sum C_i}$ on the sum of the completion times, we actually try to have the same performance ratio $\rho_{\sum C_i}$ on all the completion times.

We give below a sketch of the proof. Let us now consider that we know one of the optimal schedule for the $\sum C_i$ criterion. We can transform this schedule into a simpler but less efficient schedule as follows:

- Let C^*_{max} be the optimal makespan for the instance considered. Let k be the smallest integer such as in the $\sum C_i$ schedule considered, there is no task finishing before $\frac{C^*_{max}}{2^k}$.
- All the tasks i with $C_i < \frac{C^*_{max}}{2^{k-1}}$ can be scheduled in $\rho_{C_{max}} \frac{C^*_{max}}{2^{k-1}}$ units of time, as $\frac{C^*_{max}}{2^{k-1}}$ is the makespan of a feasible schedule for the instance reduced to these tasks, therefore bigger than the optimal makespan for the reduced instance.

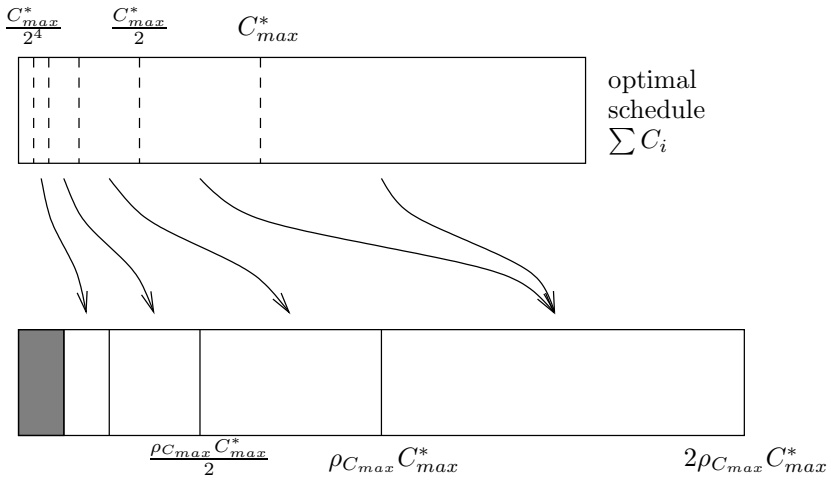

Figure 11: Transformation of an optimal schedule for $\sum C_i$ in a bi-criterion schedule (with $k = 4$).

- Similarly for $j = k - 2$ down to 1, all the tasks i with $C_i < \frac{C^*_{max}}{2^j}$ can be scheduled in $\rho_{C_{max}} \frac{C^*_{max}}{2^j}$ units of time, right after the tasks already scheduled.
- All the remaining tasks can be scheduled in $\rho_{C_{max}} C^*_{max}$ units of time, as the optimal value of the makespan is C^*_{max}. Again they are placed right after the previous ones.

The transformation and resulting schedule is shown in figure 11. If $C_i^{\mathbf{s}}$ are the completion times in the schedule before the transformation and $C_i^{\mathbf{t}}$ are the completion times after it, we can say that for all tasks i such as $\frac{C^*_{max}}{2^j} < C_i^{\mathbf{s}} \leq \frac{C^*_{max}}{2^{j-1}}$ we have in the transformed instance $\rho_{C_{max}} \frac{C^*_{max}}{2^{j-1}} < C_i^{\mathbf{t}} \leq \rho_{C_{max}} \frac{C^*_{max}}{2^{j-2}}$, which means that $C_i^{\mathbf{t}} < 4\rho_{C_{max}} C_i^{\mathbf{s}}$. With this transformation the performance ratio with respect to the $\sum C_i$ criterion is $4\rho_{C_{max}}$ and the performance ratio to the C_{max} criterion increased to $2\rho_{C_{max}}$.

The previous transformation leads to a good solution for both criteria. The last question is "do we really need to know an optimal schedule with respect to the $\sum C_i$ criterion to start with?". Hopefully the answer is no. The only information needed for building this schedule is the completion times $C_i^{\mathbf{s}}$. Actually these completion times do not need to be known precisely, as they are compared to the nearest lower rounded values $\frac{C^*_{max}}{2^j}$.

Getting these values is a difficult problem, however it is sufficient to

have a set of values such as the schedule given in figure 11 is feasible and the sum of completion times is a minimum. As the performance ratio for $\sum C_i$ refers to a value smaller than the optimal one, the bound is still valid for the optimal.

The last problem is to find a partition of the tasks into k sets where all the tasks within set S_j can be run in $\rho_{C_{max}} \frac{C^*_{max}}{2^{j-1}}$ (with algorithm $\mathcal{A}_{C_{max}}$) and where $\sum_j |S_j| \frac{C^*_{max}}{2^j}$ is a minimum. This can be solved by a knapsack with integers values.

10 Conclusion

In this chapter, we have presented an attractive model for scheduling efficiently applications on parallel and distributed systems based on Parallel Tasks. It is a nice alternative to conventional computational models particularly for large communication delays and new hierarchical systems. We have shown how to obtain good approximation scheduling algorithms for the different types of Parallel Tasks (namely, rigid, moldable and malleable) for two criteria (C_{max} and $\sum C_i$) for both off-line and on-line cases. All these cases correspond to systems where the communications are rather slow, and versatile (some machines may be added or removed at some times). Most studies were conducted on independent Parallel Tasks, except for minimizing the makespan of any task graphs in the context of off-line moldable tasks.

Most of the algorithms have a small complexity and thus, may be implemented in actual parallel programming environments. For the moment, most of them do not use the moldable or malleable character of the tasks, but it should be more and more the case. We did not discuss in this chapter how to adapt this model to the other features of the new parallel and distributed systems: It is very natural to deal with hierarchical systems (see a first study in [12]). The heterogeneous character is more complicated because most of the methods assumed the monotony of the Parallel Tasks. In the heterogeneous case, the execution time does not depend on the number of processors alloted to it, but on the set of processors as all the processors might be different.

References

[1] F. Afrati, E. Bampis, A. V. Fishkin, K. Jansen, and C. Kenyon. Scheduling to minimize the average completion time of dedicated tasks. *Lecture Notes in Computer Science*, 1974, 2000.

[2] E. Bampis, A. Giannakos, and J.-C. König. On the complexity of scheduling with large communication delays. *European Journal of Operational Research*, 94(2):252–260, 1996.

[3] J. Błażewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1996.

[4] J. Błażewicz, E. Klaus, B. Plateau, and D. Trystram. *Handbook on parallel and distributed processing*. International handbooks on information systems. Springer, 2000.

[5] J. Błażewicz, M. Machowiak, G. Mounié, and D. Trystram. Approximation algorithms for scheduling independent malleable tasks. In *Europar 2001*, number 2150 in LNCS, pages 191–196. Springer-Verlag, 2001.

[6] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, July 1974.

[7] P. Brucker. *Scheduling*. Akademische Verlagsgesellschaft, Wiesbaden, 1981.

[8] E.G. Coffman and P.J. Denning. *Operating System Theory*. Prentice Hall, 1972.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[10] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, inc., San Francisco, CA, 1999.

[11] M. Drozdowski. Scheduling multiprocessor tasks - an overview. *European Journal of Operational Research*, 94(2):215–230, 1996.

[12] P.-F. Dutot and D. Trystram. Scheduling on hierarchical clusters using malleable tasks. In *Proceedings of the 13th annual ACM symposium on Parallel Algorithms and Architectures - SPAA 2001*, pages 199–208, Crete Island, July 2001. SIGACT/SIGARCH and EATCS, ACM Press.

[13] D. G. Feitelson. Scheduling parallel jobs on clusters. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 519–533. Prentice Hall PTR, Upper Saddle River, NJ, 1999. Chap. 21.

[14] D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. *Lecture Notes in Computer Science*, 0(949):1–18, 1995.

[15] A. Feldmann, M-Y. Kao, and J. Sgall. Optimal online scheduling of parallel jobs with dependencies. In 25*th Annual ACM Symposium on Theory of Computing*, pages 642–651, San Diego, California, 1993. url: http://www.ncstrl.org, CS-92-189.

[16] I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering.* Addison-Wesley, Reading, MA, USA, 1995.

[17] M. R. Garey and R. L. Graham. Bounds on multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4:187–200, 1975.

[18] A. Gerasoulis and T. Yang. PYRROS: static scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437. ACM, jul 1992.

[19] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[20] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.

[21] D. Hochbaum, editor. *Approximation Algorithms for Np-Hard Problems.* Pws, September 1996.

[22] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[23] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[24] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. *Algorithmica*, 32(3):507, 2002.

[25] E. Jeannot. *Allocation de graphes de tâches paramétrés et génération de code.* PhD thesis, Ecole Normale Supérieure de Lyon et Ecole Doctorale Informatique de Lyon, 1999.

[26] T. Kalinowski, I. Kort, and D. Trystram. List scheduling of general task graphs under LogP. *Parallel Computing*, 26(9):1109–1128, July 2000.

[27] R. Lepère, D. Trystram, and G.J. Woeginger. Approximation scheduling for malleable tasks under precedence constraints. In *9th Annual European Symposium on Algorithms - ESA 2001*, number 2161 in LNCS, pages 146–157. Springer-Verlag, 2001.

[28] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[29] G. Mounié. *Ordonnancement efficace d'application parallèles : les tâches malléables monotones*. PhD thesis, INP Grenoble, juin 2000.

[30] G. Mounié, C. Rapine, and D. Trystram. A 3/2-dual approximation algorithm for scheduling independent monotonic malleable tasks. Technical report, ID-IMAG Laboratory, 2000. http://www-id.imag.fr/~trystram/publis_malleable.

[31] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 140–149, El Paso, Texas, 4–6 1997.

[32] M. Pinedo. *Scheduling : theory, algorithms, and systems*. Prentice-Hall, Englewood Cliffs, 1995.

[33] G. N. S. Prasanna and B. R. Musicus. Generalised multiprocessor scheduling using optimal control. In 3*rd Annual ACM Symposium on Parallel Algorithms and Architec tures*, pages 216–228. ACM, 1991.

[34] C. Rapine, I. Scherson, and D. Trystram. On-line scheduling of parallelizable jobs. In Springer verlag, editor, *Proceedings of EUROPAR'98*, number 1470 in LNCS, pages 322–327, 1998.

[35] U. Schwiegelshohn, W. Ludwig, J. Wolf, J. Turek, and P. Yu. Smart SMART bounds for weighted response time scheduling. *SICOMP: SIAM Journal on Computing*, 28, 1998.

[36] J. Sgall. Chapter 9: On-line scheduling. *Lecture Notes in Computer Science*, 1442:196–231, 1998.

[37] H. Shachnai and J. Turek. Multiresource malleable task scheduling to minimize response time. *IPL: Information Processing Letters*, 70:47–52, 1999.

[38] D.B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machine on-line. *SIAM Journal on Computing*, 24(6):1313–1331, 1995.

[39] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.

[40] B. Monien T. Decker, T. Lücking. A 5/4-approximation algorithm for scheduling identical malleable tasks. Technical Report tr-rsfb-02-071, University of Paderborn, 2002.

[41] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ, 2002.

[42] A. Tchernykh and D. Trystram. On-line scheduling of multiprocessor jobs with idle regulation. In *Proceedings of PPAM'03*, number to appear in LNCS, 2003.

[43] D. Trystram and W. Zimmermann. On multi-broadcast and scheduling receive-graphs under logp with long messages. In S. Jaehnichen and X. Zhou, editors, *The Fourth International Workshop on Advanced Parallel Processing Technologies - APPT 01*, pages 37–48, Ilmenau, Germany, September 2001.

[44] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *4th Annual ACM Symposium on Parallel Algorithms and Architectu res*, pages 323–332, 1992.

[45] J. Werglarz. *Optimization and Control of Dynamic Operational Research Models*, chapter Modelling and control of dynamic resource allocation project scheduling systems. North-Holland, Amsterdam, 1982.

[46] M.-Y. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.